


# 27

MUMPS, FORTH, PROLOG, FUTURLOG

AOUT 1986

# PROLOG

ENFIN ...

# EDITORIAL

Je ne sais si vous êtes comme moi, mais à chaque information à caratère technique diffusée par les journaux télévisés, je reste sur ma faim. Ce ne sont pas les quelques éprouvettes, les trois écrans fades et les crachotements de tuyères épisodiques qui me satisfont. Par contre, quand il s'agit d'un joueur de football quelconque, on l'interroge, le questionne. Or, le métier de footballeur consiste essentiellement à faire avancer un ballon, non à parler devant un micro. Et qu'ils sont nuls les interviews de nos tapeurs de ballon.

Je n'ai rien contre le football, mais il y a une telle disproportion entre la part faite entre l'information sportive et l'information scientifique que l'on en vient à se demander si nos chers médias craignent de nous effrayer avec des termes trop techniques. J'ai à ce propos une anecdote: nous avons été invités, notre cher Président Michel ROUSSEAU et moi-même à présenter divers usages de la micro-informatique pour en passer des séquences dans le petit journal. J'avais envisagé de présenter EXPERT-2, écrit en FORTH, sur THOMSON TO7. Une fois le matériel installé, les éclairages réglés et la caméra prête, la première question tombe. Il fallait faire simple, très simple m'a-t-on dit. Il faut qu'en trente secondes, le téléspectateur ait une idée de ce qu'est un système expert. En conclusion, EXPERT-2 est devenu un programme de diagnostic du fonctionnement du THOMSON (sinon pourquoi présenter un THOMSON...) - pardon, diagnostic c'est trop compliqué - un programme qui recherche les pannes (remarque personnelle: s'il est réellement en panne, comment fait-on marcher EXPERT-2 ?).

Prend-on réellement le public télévisé pour un âne? La télévision s'adresse à une audience très large. Si les téléspectateurs (certains...) sont aptes à retenir toutes les sélections de matchs, les divisions, les calendriers de rencontres et les compositions d'équipes (je passe les transferts et les bobos...), pouquoi ne comprendraient-ils pas un reportage sur les systèmes experts, la navigation interplanétaire ou les détail de l'acide désoxyribonucléique (un cousin du fluobenzinactivozone...). Alors j'ai décidé de me venger au travers de cet éditorial. Et, profitant de ce que nous diffusons dans ce numéro un programme en PROLOG, je vous propose une version personnelle du discours type d'un foutbaleur (il parait qu'il faut franciser, mais ils n'ont pas dit s'il faut un "l" ou deux "ll") telle qu'elle pourrait être reprise par les chroniqueurs sportifs, ce qui leur fera économiser les frais de déplacement (ce que je suis méchant...):

"lors de notre dernière rencontre en ((première, deuxième)) division lors des ((finales, demi-finales, quart de finale)) en championnat ((de france, d'écosse, d'europe)) contre ((l'Italie, la Roumanie, la Bulgarie, le Bresil, autres)), nous avons eu à faire face à ((des joueurs, une équipe, un terrain, un vent, l'altitude)) qui nous ont donné bien ((du mal, du fil à retordre, des soucis)). Mais nous espérons((la saison prochaine, le prochain match, l'année prochaine)) ((gagner, gagner, gagner, gagner))... etc...

Et avec la chaleur du mois d'août, prenez ce qui précède au second degré. Que les passionnés de football me pardonnent, la prochaine fois j'égratignerai les joueurs de tennis.

# SOMMAIRE






ASSOCIATION JEDI 8, rue Poirier de Narçay 75014 PARIS

Tel: (1) 45.42.88.90 (de 10h à 18h)

par Marc PETREMANN

### A PREMIERE VUE

Ma première rencontre avec FUTURSYS s'est produite au SICOB de printemps 1985 et c'est un peu le hasard qui m'a guidé dans cette direction. Il est vrai que le stand était très sobre, donc impossible de ne pas voir cette petite boite noire et verte.

L'idée de base du concepteur de ce système est d'apporter les notions utilisées par les systèmes experts et l'I.A.: un "pseudo-PROLOG" de poche. A cet effet, FUTURSYS est alimenté par une batterie intégrée rechargeable et reste en permanence sous tension, réglant ainsi définitivement le compte aux micro-coupures (d'habitudes ce sont elles qui ont raison de votre programme et de votre patience).

### AVEC DE LA CONFITURE

FUTURSYS est un système portable, disais-je, donc utilisable partout, y compris en voyage et par les blasés du TGV qui font PARIS LYON deux fois par semaine. Et si pris par par l'enthousiasme du problème, vous l'emmenez au wagon-restaurant, il suportera le café ou la confiture, car il est équipé d'un clavier à membrane. Bien sûr, au début, un peu habitué au ZX81 et au THOMSON, j'appuyais fortement, sur ce *)@§!*&¿ (censuré) de clavier avant de me rendre compte qu'il était plus sensible que prévu.

Enfin, FUTURSYS permet la sauvegarde des "programmes" (appelons les choses comme celà avant de vous torturer les méninges) sur un magnétophone à cassette ordinaire.

Le clavier comprend 47 touches ayant de une à quatre significations possible, et deux touches séparées RAZ et STOP situées en dehors du clavier, car dangereuses à utiliser.

Au dessus du clavier, un afficheur LCD de deux lignes de 40 colonnes permet de prendre connaissance des réactions de FUTURSYS quand on lui demande quelque chose.

### UN PETIT QUELQUE CHOSE

C'est ici que les surprises commencent. FUTURLOG qui est le langage de FUTURSYS ne sait pratiquement rien faire, au sens habituel donné en programmation, pas même une addition (dommage, moi qui ai l'habitude d'essayer mes PRINT TRUC et ECRIS CHOSE sur tout clavier qui traine, histoire de voir ce qu'il a dans les tripes). J'exagère un peu, en fait FUTURLOG sait quand même écrire:

ECRIRE23456789876543

ce qui affiche:

23456789876543

Une première constatation, FUTURLOG comprend uniquement les nombres entiers mais sans limite de taille. Ainsi, cette expression:

ECRIRE123456789012345678901234567890

sera analysée dans son intégralité.

### L'ADDITION S'IL VOUS PLAIT

L'addition, opération non disponible à la mise en route de FUTURLOG, peut cependant être définie. FUTURLOG ne faisant pas de distinctions entre les données et les instructions, à l'instar de LOGO ou LISP, développe sa connaissance sous forme de CONCEPTS eux-même partagés en STRUCTURES et en FAITS.

La définition de l'addition peut être réalisée de la manière suivante:

CS10C1RᎯPLUSᎯ.
CF10I00C1R5PLUS7=12.

et maintenant, si on tape:

ECRIRE5PLUS7

on obtient 12 à l'affichage.

Essayons:

ECRIRE3PLUS2

cà ne marche pas !!!

Eh oui, FUTURLOG ne sait additionner que 5 et 7. Pour expliquer ceci, voici l'analyse détaillée de la définition telle que nous l'avons réalisée plus haut:

CS10C1RᎯPLUSᎯ

signifie:

CS pour <u>C</u>réer <u>S</u>tructure
10 car cette Structure a le n° 10
C1 dans la base du <u>C</u>oncept 1
R dont la représentation est ᎯPLUSᎯ.

Le point et les espaces éventuels sont très importants.

CF10I00C1R5PLUS7=12

signifie:

CF pour <u>C</u>réer un <u>F</u>ait
10 de numéro 10
I00 opération <u>I</u>mplicite <u>00</u>
C1 dans la base de faits du concept 1
R5PLUS7=12 dont la représentation est celle qui suit R

Ainsi, lorsque vous tapez ECRIRE5PLUS7 , le langage FUTURSYS reconnait le fait 5PLUS7 et remplace cette expression par son membre de droite.

Déjà, vous dites-vous, çà commence mal, car apparaissent des symboles pour le moins cabalistiques. Ainsi, le signe Ꭿ qui signifie que l'on a affaire à une variable de type "Analysable Valorisable".

### DES VARIABLES DANS LES FAITS

FUTURLOG possède quatre types de variables:

Ꭿ Analysable Valorisables
Ƒ Analysables Formelles
⌐ Non Analysables Valorisables
⌐F Non Analysables Formelles

La forme des signes est en étroite relation avec leur signification. Ainsi, le signe Ꭿ est la contraction stylisée de A et v.

Intéressons-nous pour le moment au premier type de variables, Ꭿ , et voyons son rôle dans la définition de l'addition.

CS10C1RᎯPLUSᎯ.

L'utilisation de variables, dans la représentation de la structure 10 indique que tout ce qui se situera de part et d'autre de PLUS sera analysée et valorisée lors de calculs.

Mais cette valorisation n'est efficace qu'à travers une fonction implicite. Or FUTURLOG est équipé de la fonction implicite d'addition, fonction qui porte le numéro 13 et fait appel à trois arguments:

A1: numérique entier
A2: numérique entier
A9: résultat de la somme de A1 et A2.

Les expressions A1, A2 et A9 peuvent être assimilées à des "registres". Ces arguments ne peuvent en aucun cas être modifiés. Ainsi, A4+A7=A6 ne correspond pas du tout à la fonction implicite de l'addition.

On va donc redéfinir le FAIT 10, mais en faisant appel à la fonction implicite 13:

CF10I13C1RA1PLUSA2=A9.

→

ce qui signifie que nous venons de créer le Fait 10, utilisant la fonction Implicite 13 (addition) dans la base du Concept 1, dont la représentation est A1PLUSA2=A9.

Et maintenant:

ECRIRE12345678PLUS87654321

affiche

99999999

Vous pouvez additionner des nombres de 20, 30, 100 chiffres et même plus:

ECRIRE1234567801234567890l3457PLUS987654 123456789098765432

çà marchera.

Les espaces ont une importance capitale. Si on réécrit la structure 10 et le fait 10 en mettant des espaces, ceux-ci devront ensuite être utilisés dans la syntaxe FUTURLOG:

```
CS10C1RᎯ PLUS Ꭿ.
CF10I13C1RA1 PLUS A2=A9.
```

et ECRIRE3 PLUS 5    affiche 8

En fait, la manière dont une fonction implicite peut être exprimée en FUTURLOG est laissée à l'appréciation du programmeur. Voici diverses manières d'exprimer une addition:

Notation algébrique:

```
CS10C1RᎯ+Ꭿ.
CF10I13C1RA1+A2=A9.
```

Notation LOGO:

```
CS10C1RSOMMEᎯ Ꭿ.
CF10I13C1RSOMME A1 A2=A9
```

Notation FORTH:

```
CS10C1RᎯ Ꭿ +.
CF10I13C1RA1 A2 +=A9.
```

A propos de la notation algébrique, il est possible de créer la mise en parenthèses:

```
CS10C1R(Ꭿ).
CF10I0R(A1)=A1.
CS11C1RᎯ+Ꭿ.
CF11I13C1RA1+A2=A9.
```

Et maintenant ECRIRE2+3 ou ECRIRE(2)+(3) ou encore ECRIRE(2+3) sont acceptées.

De même, une opération de type ECRIRE2+3+5 sera valide. En effet, FUTURLOG analyse la chaîne de caractères de gauche à droite, et 2+3+5 est assimilé au concept Ꭿ+Ꭿ si celui-ci a été préalablement défini (bien entendu), en attribuant 2 à la variable de gauche et 3+5 à, la variable de droite:

```
Ꭿ+Ꭿ.
/  \
2   3+5
```

L'analyse isole maintenant la sous-chaîne 3+5 en appliquant la même relation, ce qui donne un arbre d'analyse:

```
Ꭿ+Ꭿ.
/  \
2   3+5
     |
    Ꭿ+Ꭿ.
    /  \
    3   5
```

Bigre ! Voilà qui ressemble à de la récursivité ! Mais voyons un cas utilisant réellement la récursivité.

### LA RECURSIVITE

Dans l'exemple suivant, on va réaliser un "programme" permettant d'obtenir la somme de n nombres entiers:

```
CS10C1RᎯPLUSᎯ.
CF10I13C1RA1PLUSA2=A9.
CS11C1RSOMME DEᎯAᎯ.
CF12I0C1RSOMME DEA1AA1=A1.
CF12I0C1RSOMME DEA1AA2=A1PLUSSOMME DEA1
PLUS1AA2.
```

Et l'utilisation de ces structures permet donc la résolution de ECRIRESOMME DE1A4

Les opérations successives seront:

```
      1PLUSSOMME DE2A4
soit: 1PLUS2PLUSSOMME DE3A4
soit: 1PLUS2PLUS3PLUS4
```

dont le résultat est 10.

Enfin, pour finir avec les exemples, voici comment créer en FUTURLOG la somme de N nom-quelconques:

```
CS10C1RᎯ+Ꭿ.
CF10I13C1RA1+A2=A9.
CS11C1RΣ(Ꭿ).
CS12C1RᎯ,Ꭿ.
CF13I0C1RΣ(A1,A2)=A1+ (A2).
CF14I0C1RΣ(A1)=A1.
```

Ainsi, quand l'argument de Σ( ) est composé de deux nombres ou plus, c'est le FAIT 13 qui est activé, sinon c'est le fait 14.

Ce qui donne pour Σ(5,7,4) la résolution suivante:

```
5+Σ(7,4)
5+7+Σ(4)
5+7+4
```

Vous pouvez aussi essayer un truc du genre:

Σ(1235,654,45,10000,1098754)

dont je ne donne pas la réponse, FUTURLOG étant plus doué que moi.

### L'INTIMITE DE FUTURLOG

En discutant avec le concepteur de FUTURSYS, j'ai appris certaines choses qui ne sont pas évidentes à la seule lecture du manuel. Ainsi, parmis certaines informations, il se trouve que le "moniteur" de FUTURLOG est écrit en FUTURLOG, y compris la manière dont les STRUCTURES et les FAITS doivent être définis:

- un détail, si on définit un FAIT ou une structure déjà existantes, tous les FAITS ou STRUCTURES voient leurs numéros d'ordre incrémentés. Par exemple, si je définis CS10C1R + . puis CS10C1R PLUS . la première version de CS10 devient CS11.

- en tenant compte de ce qui précède, on a édité les STRUCTURES et les FAITS faisant fonctionner FUTURLOG.

Sans entrer dans le détail, car çà nous mènerait trop loin, voici quelques échantillons de FUTURLOG:

le système moniteur:

```
    (F2C1) SYS=ECRIRE'OK';LIRE;TABLEAU;ATTEN
DRE;SYS.
    (F3C1) ECRIRE A1=COMMT Ø Ø Ø D A1.
    (F6C1) LIRE=COMUT Ø Ø Ø Z Ø.
    (F8C1) TABLEAU=COMTA Ø Ø Ø R Ø.
    (F9C1) ATTENDRE=COMMU Ø Ø Ø Ø Ø.
   (F10C1) COMA1 A2 A3 A4 A5 A6=A9.
   (F11C1) CFA2IA5CA1RA4= Ø .
   (F12C1) CSA3CA1RA4= Ø .
```

On voit tout de suite que le moniteur est récursif (FAIT 2, CONCEPT 1), car SYS se retrouve à droite et à gauche du terme. Pourtant FUTURLOG ne "plante" pas, car il n'y a pas d'empilement de paramètres. Ainsi, le moniteur FUTURLOG est perpétuellement ré-entrant.

Pour ce qui est de LIRE, TABLEAU, ATTENDRE et COM, les différents paramètres indiquent la manière dont l'affichage et la saisie des caractères doit être réalisée. Le signe Ø indique qu'il n'y a pas altération du paramètre prédéfini dans le système. Avec FUTURLOG, vous pouvez afficher non seulement de gauche à droite, mais aussi de droite à gauche (pour les Leonard de Vinci), mouvements de curseur compris, mais aussi de haut en bas, en diagonale, ou lettre à lettre, en fenêtre (une de saisie et une d'affichage).



CONCLUSION

FUTURLOG est un excellent outil à vocation pédagogique pour tous ceux qui s'intéressent de près à l'intelligence artificielle et aux systèmes experts. On a souvent entendu parler du projet japonais d'ordinateurs de cinquième génération et à ce titre, FUTURLOG arrive à point, donnant accès à tout le monde aux concepts qui prédomineront très certainement dans l'informatique des années futures.

FUTURSYS, par l'utilisation des variables, possède un moteur d'inférence d'ordre 1, et procède également par compilation ou interprétation, permettant une vitesse de résolution très élevée.

La possibilité de contrôler sa syntaxe, par l'intermédiaire de FAITS et de STRUCTURES permet d'adapter la formalisation des problèmes aux cas à traiter. L'utilisation de nombreux signes particuliers permet nottament une très grande concision dans l'écriture des faits et structures, ou au contraire, un glissement progressif vers une syntaxe proche de celle de langages existants tels LISP ou PROLOG. Ainsi, j'ai eu personnellement l'occasion de vérifier l'adaptabilité du contenu de "Programmez en logique avec Micro-PROLOG" (ed. EYROLLES) à FUTURLOG. Bien entendu, ma déformation de FORTHien chevroné a quelque peu altéré certaines expressions.

FUTURLOG permet d'activer le moteur d'inférence en mode "inférence multiple", ouvrant la voie au dénombrement de solutions, exploration d'arbres de possibilités, etc...

Les modes algorithmique et inférence multiple sont utilisables simultanément, permettant le calcul traditionnel, avec une très grande liberté de syntaxe, et la recherche en I.A. avec un potentiel inégalé.

Contact: toute personne désireuse d'acquérir un système FUTURSYS ou avoir des renseignements complémentaires peuvent écrire à:

INFORMATIQUE INDUSTRIE et SERVICE
BP 706
75162 PARIS CEDEX 04

Des réductions peuvent être consenties aux adhérents JEDI si plusieurs membres sont intéressés.

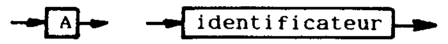
## 2ème EPISODE
## LES DIAGRAMMES SYNTAXIQUES
## REALISATION D'UN ANALYSEUR

Une seconde représentation d'une grammaire est un diagramme syntaxique, ou diagramme de Conway. Cette représentation est beaucoup plus parlante que la notation BNF, mais ces deux systèmes sont tout à fait équivalents. Les lois de représentation sont les suivantes:

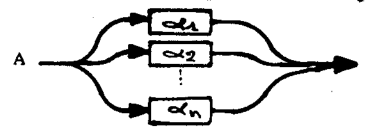
- un symbole terminal x est représenté dans un cercle ou un ovale:




- un symbole non terminal est représenté dans un rectangle:




- une production de la forme $A::=\alpha_1|\alpha_2...|\alpha_n$ est traduite par le diagramme A:




- une production de la forme $A::=\alpha_1\alpha_2..\alpha_n$ est traduite par




- une production de la forme $A::=\{\alpha\}$ est traduite par




A partir de maintenant, il est aisé de construire un analyseur syntaxique pour le langage ainsi défini; le diagramme syntaxique représente quasiment l'organigramme de l'analyseur.

Celui-ci, comme son nom l'indique, se contente d'analyser r, c'est à dire de vérifier la syntaxe des programmes. Les langages les mieux adaptés pour cela sont les langages structurés: on décompose les diagrammes syntaxiques en blocs de petites tailles à chacun desquels va correspondre une procédure.

Le PASCAL, pour exemple, convient très bien, ainsi que le FORTH. Nous aurons l'occasion d'y revenir. Pour le moment, nous utiliserons le langage naturel.

Supposons alors que nous ayons dans une variable sym le prochain symbole lu (nous travaillons en LL1), et que nous symbolisions par T(S) la procédure analysant le symbole non terminal S. Une séquence A::= s1 s2 .. sn est traduite par:

Procédure A : T(S1) ; T(S2) ; ... T(Sn) fin A

Un choix A::= S1|S2|..|Sn par, avec $l_i$=first(Si):

```
Procédure A : selon que sym l1 faire T(S1)
                        ...
                        sym ln faire T(Sn)
                        autrement erreur fin A
```

Une itération A::={S} par:

Procédure A : tant que sym L faire T(S) fin A

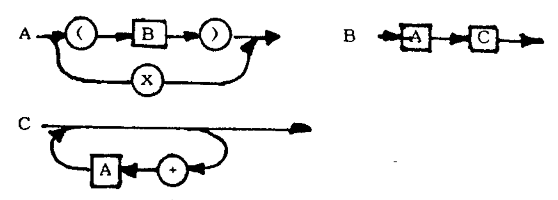
Le traitement d'un terminal x est tout simplement si sym = x alors lire symbole suivant sinon erreur.

Et le traitement d'un non terminal consiste en l'appel de la procédure correspondante.

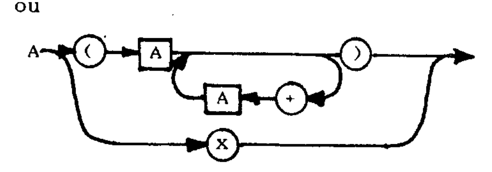
Prenons un exemple: le langage défini par

A::=x|(B)
B::= AC
C::={+A}

produisant par exemple x, (x), (x+x), ... on a:




ou




En général, sauf pour des exemples aussi simples, on préfère la première forme.

Un analyseur pour ce langage pourrait être:

```
Procédure A
 si sym = "x" alors lire sym sinon
   si sym = "(" alors
     début
       lire sym ; A ; tant que sym = "+"
         faire début
                 lire sym ; A
               fin;
       si sym = ")" lire sym sinon erreur
     fin
   sinon erreur
 fin A.
```

Et le programme est lire sym ; A fin.

Dans le prochain numéro nous passerons à la pratique. Nous définirons un véritable langage, le BLAISE, et nous écrirons un analyseur en FORTH pour ce langage.

Concevoir, écrire et mettre au point un mini-langage graphique répondant au minimum aux spécifications données ci-après, telle était l'épreuve de programmation des Olympiades de la Micro-Informatique qui se sont déroulées à NANTES en Juin 85. Le choix du langage était libre, par contre le matériel était imposé : MO5, TO7 ou TO7-70.
Le programme que je vous propose ci-après a donc été écrit à cette occasion et a obtenu la médaille d'Or. Je ne vous décris pas la tête des membres du jury losqu'ils ont su qu'il était écrit en FORTH ! Ils s'y attendaient si peu que j'ai été obligé de leur prêter ma propre cartouche FORTH ! Enfin...

Voici les spécifications minimales telles qu'elles étaient décrites pour l'épreuve, avec des erreurs volontaires.
Le MLG fonctionne sur l'environnement suivant : un écran de 25 lignes de 40 colonnes partagé en deux. La partie supérieure est le domaine graphique composé des 24 premières lignes. La partie inférieure est la ligne de commande toujours visible où l'utilisateur saisit les lignes de MLG.
Le domaine graphique est donc un espace où un pixel est défini par deux coordonnées X et Y. Avec X variant de 0 à 319 et Y variant de 0 à 191 (et non 199 comme en plein écran). Le pixel zéro (X=0,Y=0) est situé dans le coin supérieur gauche de l'écran.
L'élément de base du MLG est la ligne. Une ligne est une suite de caractères saisis par l'utilisateur à partir du clavier uniquement. Le délimiteur de fin de ligne est le caractère ASCII CR (dec 13) engendré par l'appui sur la touche ENTREE.
Dès qu'une ligne est saisie, il y a décodage et exécution IMMEDIATS de ou des instructions MLG contenues dans la ligne SANS MEMORISATION. Le MLG est un langage très restreint par rapport au Basic. Seules les constantes numériques entières sont admises. Il ne possède qu'une procédure : P.
La structure d'une instructrion MLG est la suivante :
**M** ou **Mval** ou **Mchaîne**.
**M** est un mot-clé constitué d'une seule lettre majuscule (voir liste plus loin).
**val** est une quantité entière positive ou négative dans l'intervalle -32768...+32767.
**chaîne** est une chaîne de caractères contenant une ou plusieurs instructions de MLG.
/ : le "slash" est le délémiteur standard du MLG entre chaque instruction. Il est nécessaire si une ligne comporte plus d'une instruction.
Exemples de lignes valides en MLG :

Une ligne avec une seule instruction :
N
Une ligne avec 12 instructions :
N/L/F0/C4/U159/V94/P/R50/C7/R70/C1/R90

Points particuliers pour la saisie :

En début de 25ème ligne, un caractère > indiquera à l'utilisateur que le MLG est en attente. La saisie utilisera les commodités standards des flèches d'édition, ins et eff. Une ligne aura au plus 39 caractères.

→

Vocabulaire :

**N** : nettoie l'écran sans modifier les coordonnées courantes de la plume.
**O** : origine. Met les coordonnées courantes à zéro.
**M** : milieu. Met les coordonnées courantes à U=159 et V=94.
**B** : baisse la plume de façon à laisser une trace sur l'écran et donc dessiner.
**L** : lève la plume pour se déplacer sans laisser de trace.
**Cval** : établit la couleur de trace à la valeur val. (val appartient à 0..7).
**Fval** : établit la couleur de fond à la valeur val.
**Xval** : actualise la coordonnée X à X + val.
**Yval** : actualise la coordonnée Y à Y + val.
**D** : se déplace jusqu'au point courant (avec ou sans trace suivant L et B).
**Uval** : actualise X à la valeur val.
**Vval** : actualise Y à la valeur val.
**Rval** : trace un cercle (rond) de rayon val et de centre ayant pour coordonnées les coordonnées courantes.
**Pchaîne** de caractères : définit la procédure courante qui est mémorisée en lieu et place de celle établie par une instruction P antérieure.
**E** : active (exécute) la procédure courante.

Exemples :

N/O/M/F0/C4/R50/C7/R70/C1/R90

Efface l'écran, se place au milieu de l'écran, met le fond en noir, la trace en bleu, dessine trois cercles concentriques bleu, blanc et rouge, de rayons respectivement 50, 70 et 90 pixels.

PB/X50/Y50/X-50/Y-50

Définit la procédure courante comme le tracé plume baissée d'un carré de 50 pixels de côté.

L/O/P/L/M/P

Lève la plume, se place à l'origine, trace le carré, se place au milieu et trace de nouveau le carré.

```
SCR: 199
: SI ;
: ALORS [COMPILE] IF ; IMMEDIATE
: SINON [COMPILE] ELSE ; IMMEDIATE
: FINSI [COMPILE] THEN ; IMMEDIATE
: TANTQUE [COMPILE] BEGIN ; IMMEDIATE
: FAIRE [COMPILE] WHILE ; IMMEDIATE
: FINTANTQUE [COMPILE] REPEAT ; IMMEDIAT
E
: REPETER [COMPILE] BEGIN ; IMMEDIATE
: JUSQUA ;
: FINREPETER [COMPILE] UNTIL ; IMMEDIATE


VARIABLE FD         VARIABLE CC
VARIABLE X          VARIABLE Y
VARIABLE BAISSE     VARIABLE val
VARIABLE COMPT      VARIABLE LONG
VARIABLE XP         VARIABLE YP
VARIABLE COMPT1     VARIABLE LONG1
VARIABLE MOT        6 STRING CHAINE
40 STRING ENTREE    40 STRING AUX
```

```
SCR: 200
: FENETRE 24 24 WINDOW ;

: FOND 0 23 WINDOW GLOBAL PAPER
  FD @ DUP COLOR GCOLOR ;

: COULEUR LOCAL INK CC @ GCOLOR ;

: ORIGINE 0 X ! 0 Y ! ;

: MILIEU 159 X ! 94 Y ! ;

: XVAL val @ X +! ;

: YVAL val @ Y +! ;

: UVAL val @ X ! ;

: VVAL val @ Y ! ;

: NETTOIE FOND CLS COULEUR ;
```

```
SCR: 201
91 ARRAY SINTABLE
: EMPLIR DO I SINTABLE ! LOOP ;
10000 9998 9994 9986 9976 9962 9945 9925
 9903 9877 9848 9816 9781 9744 9703 9659
 9613 9563 9511 9455 9397 9336 9272 9205
 9135 9063 8988 8910 8829 8746 8660 8572
 8480 8387 8290 8192 8090 7986 7880 7771
 7660 7547 7431 7314 7193 7071 6947 6820
 91 43 EMPLIR
 6691 6561 6428 6293 6157 6018 5878 5736
 5592 5446 5299 5150 5000 4848 4695 4540
 4384 4226 4067 3907 3746 3584 3420 3256
 3090 2924 2756 2588 2419 2250 2079 1908
 1736 1564 1392 1219 1045 0872 0698 0523
 0349 0175 0000
 43 0 EMPLIR FORGET EMPLIR
: >180 DUP |SI 90 > ALORS 180 SWAP -
           |FINSI   SINTABLE @ ;
: SIN 360 MOD DUP |SI 0< ALORS 360 +
                  |FINSI
  DUP SI 180 > ALORS 180 - >180 NEGATE
      |         SINON >180
    FINSI ;
: COS 90 + SIN ;


SCR: 202
: DEPLACE
  SI BAISSE @ ALORS X @ Y @ LINETO
  |           SINON PAPER X @ Y @ PSET
  |                 INK
  FINSI ;

: CERCLE
  val @ X @ + Y @ PSET 361 1
  DO
  | X @ val @ I COS 10000 */ +
  | Y @ val @ I SIN 10000 */ + LINETO
  LOOP
  PAPER X @ Y @ PSET INK ;

: VISU FENETRE GLOBAL PAPER 0 COLOR
  INK 3 COLOR 0 24 LOCATE 40 SPACES
  BELL INVCOLOR ;

: SAISIE VISU CSRON
  0 24 LOCATE ." >" 39 LINPUT$ ENTREE $!
  " /" ENTREE $+ FOND COULEUR CSROFF ;

: TEMPO 3000 WAIT ;


SCR: 203
: CONV MID$ DROP 1- CONVERT DDROP ;

: CONVERSION
  SI CHAINE LEN 1 > ALORS
  | SI CHAINE 2 1 MID$ " -" $=
  | |  ALORS 0. CHAINE 3 CHAINE LEN 2-
  | |       CONV NEGATE
  | |  SINON 0. CHAINE 2 CHAINE LEN 1-
  | |       CONV
  | FINSI
  | val !
  FINSI ;

: VIDE "         " CHAINE $! ;

: INIT 1 FRAME 0 FD ! 3 CC ! NETTOIE
  0 BAISSE ! ORIGINE DEPLACE ;

: AFFICHE VISU
  0 24 LOCATE ." Pt:" X ? ." ; " Y ?
  ." Coul.:  " CC ? ." Plume: "
  SI BAISSE @ ALORS ." Baissee"
  |            SINON ." Levee"
  FINSI TEMPO FOND COULEUR ;


SCR: 204
: ERREUR BELL BELL ;
: CHAINE? CHAINE TYPE TEMPO ;
: SORTIE
  ERREUR VISU ." Sortie ecran dans "
  CHAINE? ;
: TESTX SI X @ DUP 0< SWAP 319 > OR
        | ALORS SORTIE XP @ X !
        FINSI ;
: TESTY SI Y @ DUP 0< SWAP 191 > OR
        | ALORS SORTIE YP @ Y !
        FINSI ;
: TESTCERCLE
  SI val @ >R Y @ DUP R@ - 0< SWAP R@ +
  |  191 > OR X @ DUP R@ - 0< SWAP R> +
  |  319 > OR OR ALORS SORTIE
  |                SINON CERCLE
  FINSI ;
: SYNTAXE ERREUR VISU ." Erreur de synta
xe dans " CHAINE? ;
: TESTC val @ DUP 0< SWAP 7 > OR ;
: ERC ERREUR VISU ." Couleur illegale da
ns " CHAINE? ;


SCR: 205
: EXECUTION CHAINE ASC CASE
78 OF NETTOIE ENDOF
79 OF ORIGINE DEPLACE ENDOF
77 OF MILIEU  DEPLACE ENDOF
66 OF 1 BAISSE ! ENDOF
76 OF 0 BAISSE ! ENDOF
67 OF CONVERSION SI TESTC ALORS ERC
SINON val @ CC ! COULEUR FINSI ENDOF
70 OF CONVERSION SI TESTC ALORS ERC
SINON val @ FD ! FOND FINSI ENDOF
88 OF CONVERSION X @ XP ! XVAL TESTX
      ENDOF
89 OF CONVERSION Y @ YP ! YVAL TESTY
      ENDOF
85 OF CONVERSION X @ XP ! UVAL TESTX
      ENDOF
86 OF CONVERSION Y @ YP ! VVAL TESTY
      ENDOF
68 OF DEPLACE ENDOF
82 OF CONVERSION TESTCERCLE ENDOF
69 OF MOT @ EXECUTE ENDOF
83 OF INITSCR INK 2 COLOR ABORT ENDOF
65 OF AFFICHE ENDOF
SYNTAXE              ENDCASE ;


SCR: 206
: DECODAGE
SI ENTREE ASC 80 =
|  ALORS ENTREE 2 ENTREE LEN 1- MID$
|        AUX $!
|  SINON 1 COMPT !
|        TANTQUE COMPT @ ENTREE LEN <
|        |      FAIRE 0 LONG !
|        |      TANTQUE ENTREE COMPT @
|        |      |     LONG @ + 1 MID$
|        |      |     " /" $= NOT
|        |      |     FAIRE LONG 1+!
|        |      FINTANTQUE
|        |      VIDE ENTREE COMPT @
|        |      LONG @ MID$ CHAINE $!
|        |   SI CHAINE ASC 80 =
|        |   |  ALORS ENTREE COMPT @ 1+
|        |   |        ENTREE LEN COMPT @
|        |   |        - MID$ AUX $!
|        |   |        ENTREE LEN COMPT !
|        |   |  SINON EXECUTION COMPT @
|        |   |        LONG @ + 1+ COMPT !
|        |   FINSI
|        FINTANTQUE
FINSI ;
```

THOMSON

Suite page 12

# XV LES VARIABLES GLOBALES

## A) Introduction

Depuis le début de ce petit manuel, nous n'avons utilisé que des variables mémoire ou "variables locales". Ces variables sont à la disposition de l'utilisateur, et de lui seul, uniquement pendant sa session. Elles ont une duree de vie limitée. Bien que nous ayons affirmé : MUMPS ne travaille jamais sur des fichiers. Il est, malgré tout, important de pouvoir conserver les informations soit entrées, soit calculées. Ce sera le rôle des variables "globales", qui se différencient des variables locales par le fait qu'elles sont accessibles à plusieurs utilisateurs (s'ils en ont l'autorisation). Les utilisateurs familiers avec des langages tel que APL et ALGOL ou PASCAL devront utiliser le terme "local" avec précautions.

## B) Les variables GLOBALES

Les variables globales répondent aux règles enoncées pour les variables locales à la différence qu'elles sont précédées par un accent circonflexe. Elles pourront être indicées. Lorsqu'on fait référence, à ce qu'on a l'habitude d'appeler un fichier, en MUMPS on utilise une arborescence de variables globales. Le contenu des variables globales peut etre affecté à des variables locales.

Attention ! Il existe trois situations dans lesquelles les variables globales NE DOIVENT PAS ETRE UTILISEES :

- Comme variable de contrôle de boucle (avec l'ordre FOR)

- Comme variable réceptrice dans une commande READ

- Comme argument entre parenthèses de la commande KILL

## C) Reference simplifiée des variables globales (Naked references)

MUMPS permet, en plus de ce que nous avons déja vu, d'utiliser une forme simplifée pour référencer les variables globales. Cette forme permet de citer un ou plusieurs index (indices) sans avoir a redéfinir le nom et les indices précédemment utilisés. Examinons la commande suivante :

```
SET ^A(1,2,3)=3,^A(1,2,4)=4,^A(1,2,5)=5
```

En utilisant la forme simplifiée nous pouvons écrire la même commande de la facon suivante :

```
SET ^A(1,2,3)=3,^(4)=4,^(5)=5
```

Dans cette ligne de commande nous avons :

- Alimente la globale ^A(1,2,3) avec 3

- " " " référencée par la derniere "naked reference" en y ajoutant l'indice 4

- " " " référencée par la meme "naked reference" en y ajoutant l'indice 5

En fait MUMPS a maintenu dans la "naked reference" la description : ^A(1,2 .

NOTE IMPORTANTE : MUMPS conserve toujours la description de la dernière variable citée moins le dernier indice se trouvant près de la parenthèse fermante. La naked référence peut être erronée lorsque :

- l'utilisateur n'a jamais fait appel à une variable gobale
- apres l'utilisation d'une globale sans indice
- ou à la suite d'un $DATA ou un $NEXT quand le niveau retourné n'existe pas

→

La naked référence permet également de citer plusieurs indices. Exemple :

```
SET ^A(1,2)=7,^(2,3)=15
```

Ceci est identique à : SET ^A(1,2)=7,^A(1,2,3)=15

Un dernier exemple illustre l'utilisation de la naked référence. Imaginons le traitement suivant : Apres avoir verifié de l'existence de la variable ^A(I), nous voulons créer 100 nouvelles variables du type ^A(I,J)=J . Nous pouvons écrire :

```
IF $DATA(^A(I)) FOR J=1:1:100 SET ^(I,J)=J
```

Apres l'exécution de cette ligne l'arborescence sera composée de 101 indices, puisqu'à chaque passage dans la boucle la naked référence est incrémentée d'un indice. Le résultat global de l'arborescence est :

```
^A
^A(I)
^A(I,1)=1
^A(I,I,2)=2
^A(I,I,I,3)=3
     //
     //
^A(I,I,I,I,I,I,I,I,I,I....,I,I,I,I,I,I,100)=100
```

Si nous avions voulu avoir une arborescence verticale à la place d'une arborescence horizontale, nous aurions écrit :

```
IF $DATA(^A(I)) SET ^(I,1)=1 FOR J=2:1:100 SET ^(J)=J
```

Regardons la ligne suivante et commentons-la.

```
SET ^A(1),^(2)=^(3,4)
```

Quand nous affectons le contenu d'une variable à une autre variable, MUMPS commence par vérifier que la variable emmettrice existe (c'est également vrai pour les variables locales). Par conséquent, le fait d'écrire :

```
SET ^A(3,4)="TATA",^A(1)=1,^(2)=^(3,4)
```

Permet d'affecter à ^(2) le contenu de ^(3,4). Mais en réalite l'affectation est S ^A(3,2)=^A(3,4) puisque la naked référence est d'abord ^A(3 puis ^A( et ensuite ^A(3.

D) La commande LOCK

Dans un environnement multi-utilisateurs, les différentes taches actives doivent être synchronisées. La synchronisation est nécessaire afin de prévenir les incohérences ou les dégradations de la base de données, lors d'accès simultanés par plusieurs utilisateurs aux mêmes informations. Il est impensable qu'un usager introduisant un nouveau document autorise son voisin à le supprimer dans le même temps. L'utilisation de la commande LOCK évite ce désagrement. Cette commande est conventionnelle, pour qu'elle soit effective, il est important qu'on la retrouve à chaque fois que l'on utilise une ressource. Pour imager cette commande, imaginons deux personnes ayant chaqu'une une clef pour ouvrir une porte. La première personne ayant introduit sa clef dans la serrure interdira l'accès à la deuxième. Nous pouvons compliquer le problème en disant qu'une des personne veut ouvrir et l'autre fermer la porte. Le status de la porte dépendra en finalité de la dernière personne y ayant eu l'accès. C'est ce phénomène que les programmeurs "MUMPS" doivent prendre en compte. Regardons la routine ci-dessous et immaginons que deux utilisateurs l'exploitent au même instant. Qu'est ce qui va se passer ??

```
MAJCLI    ;MISE A JOUR CLI   Y.L.G.   2/NOV/84   3/NOV/84
          S NBCLI=^CLI(0)+1
          R "entrez le nom du nouveau client : ",NOM
          S ^CLI(NBCLI)=NOM,^CLI(0)=NBCLI
```

→

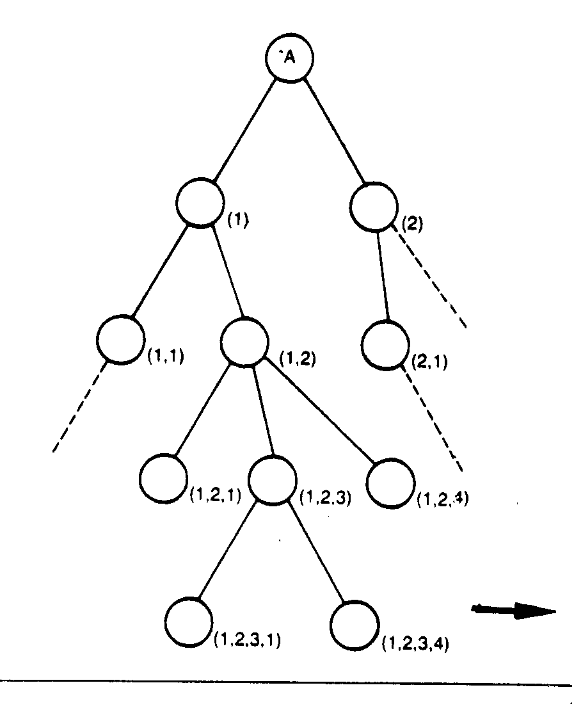
Les deux utilisateurs auront chacun la même valeur dans NBCLI, le contenu de ^CLI(NBCLI) après l'exécution du deuxième ordre SET contiendra le dernier nom saisi, le premier étant remplacé par le second. La commande LOCK ou L résoud ce délicat problème.

En effet, cette commande permet de réserver des ressources logiques. Elle s'applique sur les variables locales ou globales. Elle verrouille toute l'arborescence située en dessous, appelée "descendants", de la ou des variables citées, mais également les "ascendants" (voir figure ci-apres). Un nouvel ordre LOCK déverrouille les ressources précédemment verouillées. Cette fonctionnalité est très pratique car elle évite les situations de "dead lock" (étreinte mortelle).Cette situation survient dans d'autres systèmes lorsqu'un utilisateur a verrouillé la ressource X, un deuxième verrouille la ressource Y et demande la ressource X alors que le premier demande la ressource Y. Réecrivons la routine ci-dessus, en prenant soin de verrouiller la variable NBCLI. Si le même programme est appelé, il attendra que la ressource NBCLI soit liberée.

```
MAJCLI   ;MISE A JOUR CLI  Y.L.G.  2/NOV/84  3/NOV/84
         LOCK NBCLI S NBCLI=^CLI(0)+1
         R "entrez le nom du nouveau client : ",NOM
         S ^CLI(NBCLI)=NOM,^CLI(0)=NBCLI LOCK
```

Un temps d'attente maximum peut être ajouté aux arguments de la commande LOCK. Si ce temps est depassé, la variable $TEST est, comme d'habitude, alimentée avec la valeur 0 (faux) et aucun verouillage n'est effectué. La syntaxe du LOCK n'étant pas très simple à expliquer nous allons voir et commenter quelques exemples :

| Commande | Resultat |
|---|---|
| LOCK | déverrouille toutes les variables précédemment verouillées |
| LOCK ^A | verrouille la variable ^A et tous ses descendants |
| LOCK A,B | verrouile A, deverrouille A, verrouille B |
| LOCK (^A,B) | verrouille B et ^A plus ses descendants |
| LOCK A:2 | verrouille A dans un temps limité de deux secondes. $TEST contiendra: 1 (vrai) si A est verrouillée, 0 (faux) dans l'autre cas |
| LOCK ^A(1,2,3) | verrouillé: ^A(1,2,3) ^A(1,2,3,4) ^A(1,2,3,1) mais aussi: ^A(1,2) ^A(1) ^A |




ARBORESCENCE DE LA VARIABLE ^A

**RESUME**

Dans ce chapitre nous avons vu ce qu'étaient les fichiers en MUMPS. Ce ne sont que des variables stockées et partageables. Difference entre une variable locale et une variable globale : l'accent circonflexe. Dautre part nous avons vu qu'il était possible de gérer les conflits d'accès aux variables. Il est bon de rappelerque les variables locales sont seulement disponnibles pendant la session de connexion avec la machine. Avec toutes les notions déja vues, il est facile de percevoir la puissance et la simplicité du langage MUMPS dans la gestion des bases de données. ■


*Suite de la page 8*

```
SCR: 207
: PROCEDURE
1 COMPT1 !
SI AUX ASC 47 =
 | ALORS COMPT1 1+!
FINSI
TANTQUE COMPT1 @ AUX LEN < FAIRE
    |   0 LONG1 !
    |   TANTQUE AUX COMPT1 @ LONG1 @ + 1
    |      |    MID$ " /" $= NOT
    |      |    FAIRE LONG1 1+!
    |   FINTANTQUE
    |   VIDE
    |   AUX COMPT1 @ LONG1 @ MID$ CHAINE
    |   $! EXECUTION
    |   COMPT1 @ LONG1 @ + 1+ COMPT1 !
FINTANTQUE ;

' PROCEDURE CFA MOT !
```

```
SCR: 208
: rnd 7 RND 1+ ;
: ALEA rnd COLOR (EMIT) rnd 52 + NOTE ;
: EMIT1 ' ALEA CFA 1 SYSVEC ! ;
: EMIT2 ' (EMIT) CFA 1 SYSVEC ! ;
: PRESENTATION VISU INVCOLOR
  0 24 LOCATE ." Appuyez sur une touche
pour continuer" FOND COULEUR
" B/X47/D/Y47/D/X-47/D/Y-47/D/" AUX $!
" E/L/U48/V48/D/C5/E/L/U96/V96/D/C6/E/"
ENTREE $!
DECODAGE EMIT1 2 DUR CSROFF
REPETER 1 1 SIZE 2 3 LOCATE ." M"
    |   8 9 LOCATE ." L"
    |   14 15 LOCATE ." G" 0 0 SIZE
    |   25 18 LOCATE ." J.P.POSTEC"
    |   JUSQUA ?TERMINAL
FINREPETER
EMIT2 24 DUR INIT ;

: MLG INIT PRESENTATION
  BEGIN
   |   SAISIE DECODAGE
  AGAIN ;
```

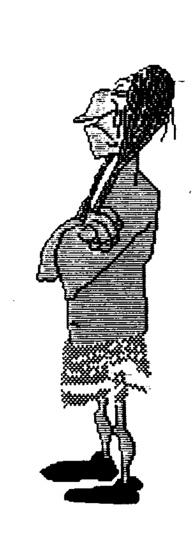
<u>Quelques explications à propos de ce programme</u> :

-L'écran 199 contient la définition des mots de structure afin de pouvoir utiliser un langage proche de l'algorithmique. Il contient également les déclarations des diverses variables utilisées.

-L'écran 201 est relatif à la trigonométrie inexistante dans le FORTH de base du TO7-70. Cela est nécessaire pour le tracé des cercles.

-L'écran 204 contient tous les tests (sortie d'écran, syntaxe, couleur). Cette partie n'était pas demandée dans l'épreuve mais a été ajoutée pour le confort de l'utilisateur.

-L'écran 205 se rapporte à l'exécution de la chaîne selon la lettre de début. Il est à noter que j'ai ajouté 2 mots, toujours pour le confort :
A : affiche les coordonnées X et Y du point courant, le numéro de la couleur courante ainsi que l'état de la plume : Baissée ou Levée.
S : permet de sortir (proprement!) du programme.



-L'écran 206 est relatif au décodage de la ligne entrée puis à son exécution.

→

-L'écran 207 contient le mot relatif au décodage et à l'exécution de la procédure. Celle-ci est sauvegardée dans la variable AUX lors du décodage général (écran 206). Vous pouvez remarquez que le CFA de PROCEDURE est stocké dans la variable MOT. Ceci est à lier au mot EXECUTION qui lorsque le code ASCII de la première lettre est 69 (E) exécute ce qui est à l'adresse que contient MOT, donc exécute PROCEDURE. Ceci s'appelle une indirection et permet dans certains cas de définir des mots après que l'on en ait besoin. A rapprocher des procédures "FORWARD" de PASCAL!

-L'écran 208 contient le mot PRESENTATION qui utilise lui-même le MLG avec une procédure. Remarquez la vectorisation avec les mots EMIT1 et EMIT2. Le mot MLG est le programme principal et c'est lui qu'il faut taper pour lancer le jeu.

Faites de beaux dessins !

# Les micro-ordinateurs FORTH de Rockwell

par B.DEREGNAUCOURT

Realiser a peu de frais un micro-ordinnateur programmable en forth est devenu possible depuis la sortie sur le marche de 2 microprocesseurs monochip le R65F11 et le R65F12. Rockwell a developpe autour de ces cicuits integres et du R65010 plusieurs configurations de micro qui integrent toutes le Rockwell Single Chip (RSC) FORTH.

Sans aborder le cote technique,nous allons decrire les possibilites de ces systemes

- une UC 6502 avec 4 nouvelles instructions : SMB,RMB,BBS,BBR.
- 192 octets de RAM
- pour les entrees/sorties on dispose de ports paralleles 8 bits
  - 2 ports pour le R65F11
  - 5 ports pour le R656F12
  - 4 ports pour le R65010
- 2 compteurs/timers programmables 16 bits avec latchs
- une interface serie asynchrone full duplex de 50 a 9600 bauds ( sans gestion de lignes de controle )
- 10 sources d'interruptions possibles :
  - 4 lignes a detection de front (2 positives,2 negatives)
  - une entree RESET
  - une entree NMI
  - 2 compteurs 16 bits
  - les donnees recues ou transmises sur la liaison serie
- 2 vitesses sont possibles 1 Mhz ou 2 Mhz

Ces caracteristiques sont communes aux configurations envisageables :

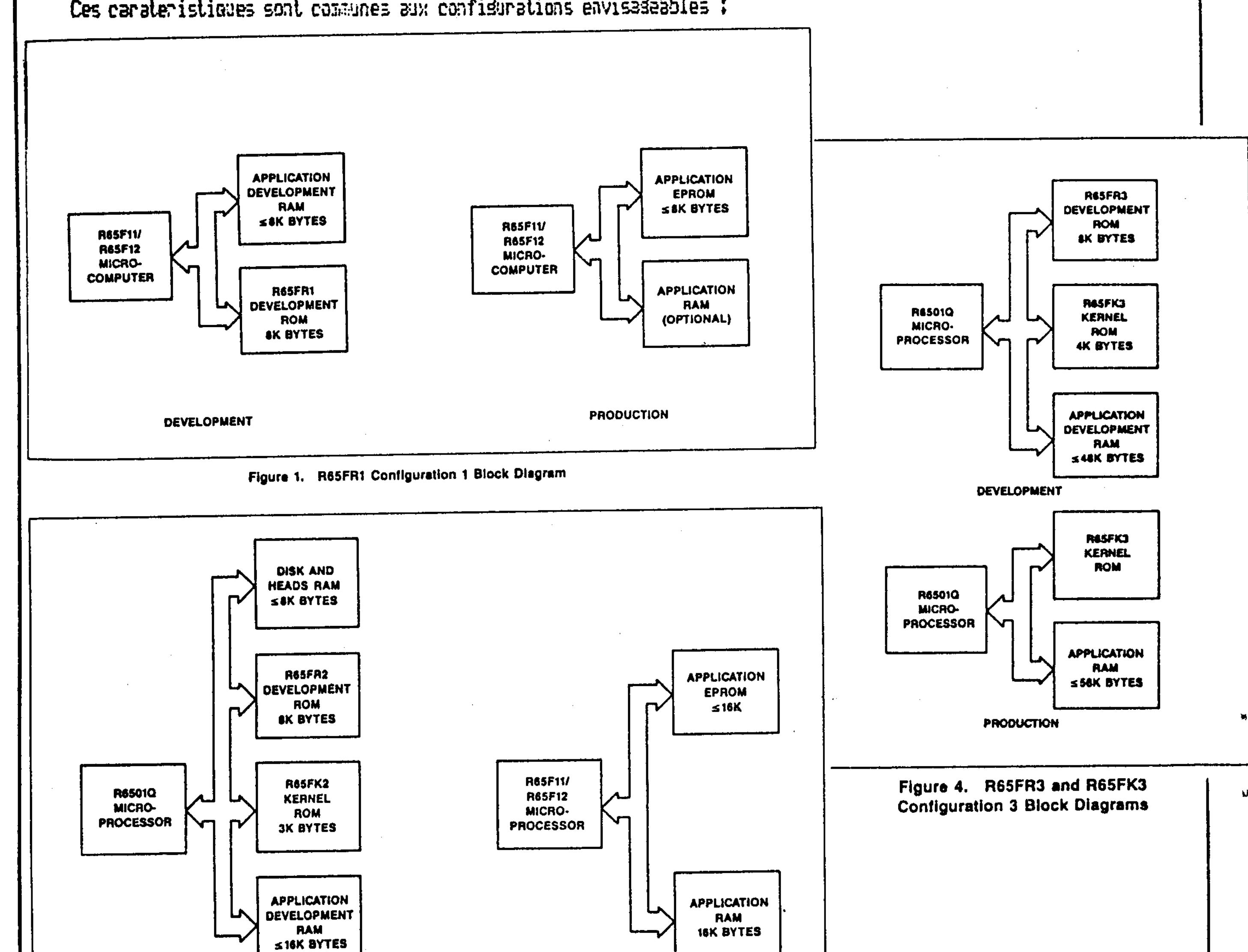



Figure 1. R65FR1 Configuration 1 Block Diagram

Figure 4. R65FR3 and R65FK3 Configuration 3 Block Diagrams

Figure 2. R65FR2 and R65FK2 Configuration 2 Block Diagrams

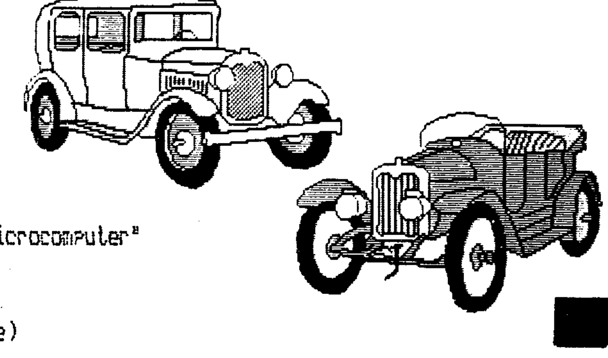
A tous ces systemes, on peut ajouter une interface lecteur de disquettes.Nous avons utilise le WD2793 qui peut commander n' importe quel lecteur 5 ou 8 pouces, simple ou double face, 35, 40, ou 80 pistes.
Pratiquement, une seule configuration est realisable (la premiere), elle a ete longuement etudiee dans la revue Micros et Robots et reprends les shemas proposes par Rocwell en y adjoignant une interface controleur de disque.
Le circuit imprime est disponible chez FACIM.pour les autres configurations, il n' existe pas de circuit imprime en France et nous n' avons pas trouve de documentations (meme chez Rockwell France).

Pour la console,nous avons utilise un minitel via la liaison RS232 du micro et la prise peri-informatique du minitel moyennant une petite interface d'adaptation. Neanmoins la prom de developpement necessite d' etre legerement modifiee car le micro initialise la liaison RS232 en 7 bits sans parite alors qu' il faut 7 bits parite paire pour le minitel
Mais si vous possedez deja un ordinnateur muni d' une liaison RS, rien ne vous empeche d' utiliser la console, l' ecran et le drive de cet ordinnateur, le RSC-FORTH autorisant la compilation directe par liaison RS232 !
Tout cela donne une configuration minimum a 1500F environ. (le circuit R65F11 est distribue par la societe Ern).
L'adjonction de l' interface pour lecteur de disquettes coute environ 500F, prix auquel il faut rajouter celui du ou des lecteurs.

Quand au Forth propose, il est tres complet (sur-ensemble du Fig Forth 79 V1.0) la rom de developpement contient aussi un moniteur et un assembleur. Le manuel livre avec la rom est extremement bien fait, c' est presque un cours de forth !
Pour les applications industrielles, rappellons que le code compile de ce forth est extremement compact meme compare au langage machine, on peut compiler avec entetes separes et reloger le code en RAM, PROM ou ROM. Le programme mis en rom devient auto-start (pas de boot a faire) simplement en executant auparavant un mot particulier (AUTOSTART), les entetes en sont supprimees et son execution en mode production ne necessite plus la rom de developpement mais seulement une version reduite du noyau forth contenu dans ce qu' on appelle la Kernel rom (rom masquee du micro).
Avec ses 48 lignes d' entrees/sories, ses deux compteurs/timers programmables et son tres faible encombrement, il permet de resoudre bien des problemes ou ses concurrents ne propose que l' assembleur.

ADRESSES UTILES

FACIM, 19 rue de Hegenheim, 68300 SAINT LOUIS
ERN, 237 rue de Fourny, ZAC de BUC, 78530 BUC



REFERENCES

MICRO ET ROBOTS N 11 et 12 (Oct. et Nov. 84)
NOTE D' APPLICATION ROCWELL N 2162 :
" A Low-Cost Development Module For The R65F11 FORTH Microcomputer"

RENSEIGNEMENTS

JEAN-CLAUDE LEMAIGRE Tel (16 1) 60 86 39 84 (domicile)

# FORTH Extraction de racine

par Marc PETREMANN

```
: SQRT ( n --- racine carrée de n )
DUP 0< ( test si n est inf à 0 )
IF  ." Argument illégal" ABORT
ELSE DUP 2/
    10 0
    DO
        DDUP / + 2/
    LOOP
    SWAP DROP
THEN. ;
```

Ce programme calcule la racine carrée d'un nombre n selon le principe de l'algorithme de NEWTON. Il ne s'applique qu'aux nombres entiers de 16 bits et ne délivre que la partie entière de la racine. Le résultat obtenu est équivalent à l'opération INT(SQR(n)) en langage BASIC.

Exemples:

```
81 SQRT
( itérations: 21 12 9 9 9 9 9 9 9 9 )  . affiche 9

32761 SQRT
( itérations: 8191 4097 2052 1033 532 296 203 182 181 181 )
  . affiche 181
```

Les valeurs délivrées par 1 SQRT et 0 SQRT sont erronnées. Un test est effectué si on cherche à extraire la racine carrée d'un nombre négatif.

Le procédé d'obtention de la racine carrée par ce procédé est certes discutable, mais il offre l'avantage d'être assez rapide d'exécution et s'applique fort bien aux applications graphiques où l'on se passe des valeurs fractionnaires (si vous avez des demi pixels, c'est que vous êtes dotés d'un interface graphique révolutionnaire).

Cet algorithme peut commodément être réécrit en assembleur ou en langage PASCAL (utilisation des INTEGER). Bien entendu, si vous étendez ce domaine aux entiers double précision, il faudra augmenter le nombre d'itérations (boucle DO..LOOP), ce qui diminuera les performances de la définition actuelle.

```
list_banque(diag).
cons(q.*1)
    -ajtb(item(*1)."O")
    -//.
cons(*1)
    -lirte(*2)
    -cons(*2.*1).

dans_l(*1.*2,*3)
    -dans_ou(*1,*3)
    -dans_l(*2,*3).
dans_l("O",*3)
    -//.

equival(*1,*2)
    -dans_l(*1,*2)
    -dans_l(*2,*1).

chercher
    -cls
    -ln
    -outm(" ****  Entrez vos items,suivis d'un pt_virgule  ****")
    -ln
    -outm(" ****   tapez q et pt_virgule pour finir  ****")
    -ln
    -prompt(".")
    -cons("O")
    -prompt(">").
chercher
    -ln
    -i_v
    -init_ecr
    -v_n
    -window(4,18,1,54)
    -ln
    -compar.
chercher
    -window(1,10,1,50)
    -ln
    -ln
    -ef_paq(item)
    -ef_paq(item_non)
    -ajtb(item_non("O")."O")
    -outm(" ****  session terminee  ****")
    -ln
    -fin_ses.

sous_l(*1.*2,*3,*4)
    -non(dans_ou(*1,*3))
    -sous_l(*2,*3,*1.*4).
sous_l("O",*1,*4)
    -ajtb(ss_lis(*4)."O")
    -//.
sous_l(*1.*2,*3,*4)
    -dans_ou(*1,*3)
    -sous_l(*2,*3,*4).

compar
    -diag(*1,*2)
    -compar(*1,*2).
compar(*1,*2)
    -item_non(*3)
    -item_ds(*2,*3)
    -window(23,24,32,60)
    -outt(*1)
    -//.
compar(*1,*2)
    -item(*3)
    -dans_l(*3,*2)
    -window(19,20,32,60)
    -outt(*1)
    -ln
    -sous_l(*2,*3,"O")
    -question(*1)
    -ln.
```

Ecrit en CRIL-PROLOG, avec modification des noms de primitives conformément au Prolog dit "de Marseille".

Il s'agit de traiter des informations ayant comme objectif de faire coincider un diagnostic avec une liste d'items sensés représenter ce même diagnostic.

Les clauses représentant un diagnostic seront de la forme
diag(*1,*2).
diag étant la tête de clause
*1 étant le nom générique du diagnostic
*2 la liste des items constituant ce diagnostic, terminée par "O", équivalent de nil en Pascal.
Ex: diag(GRIPPE,fièvre.courbature.asthénie..."O").

On utilise également deux autres listes:
- -item(.........."O") étant la liste des items confirmés de façon absolue soit dès l'entrée, soit à la suite de la phase de questionnement.
- -item_non(.........."O") étant la liste des items infirmés de façon absolue...

Le programme est hiérarchisé en plusieurs procédures principales:
- -chercher.
- -compar.
- -question.

et des procédures utilitaires travaillant sur les listes:
- -cons. qui construit une liste "item(......)".
- -dans_ou. qui recherche si une sous-liste est dans une liste.
- -dans_l. qui recherche une liste dans une autre.
- -equival. qui recherche l'équivalence de deux listes, les items (ou termes) n'étant pas forcément dans le même ordre.
- -item_plus., item_ds., terme_plus. travaillant sur l'ajout et la recherche de termes dans une liste.

et des procédures d'affichage écran propre à l'APRICOT PC (nottament les procédures de fenètrage) permettant d'avoir l'écran suivant:

```
--------------------------    --------------------------
+  procedure question     +   +    liste des items     +
+                         +   +                        +
+  items x valide ?       +   +                        +
+                         +   + valides     invalidés  +
+  O/N ?                  +   +                        +
+                         +   +                        +
--------------------------    --------------------------

          --------------------------
          + diagnostic possible: x +
          +                        +
          + diagnostic confirme: x +
          +                        +
          + diagnostic exclu   : y +
          +                        +
          --------------------------
```

Le programme démarre en effaçant le paquet de clauses "chercher".

1-entrées des items (ou symptomes repérés), ce qui produit une liste "item(.........."O").

2-le programme cheche à effacer la première clause du paquet "diag" en utilisant les clauses "compar":
-si une liste d'item est approchante de la liste de symptômes dans ce diag(a,liste)

-a il cherche en premier l'existence de "contre-indications" contenue dans la liste "item_non", ce qui lui fait sauter à une autre clause "diag" en cas d'effacement.

-b il essaie de compléter la liste "item" si nécessaire en utilisant les clauses questions, ainsi que la liste item_non.

-c arrivé à un résultat, il l'affiche et passe à la clause "diag" suivante, jusqu'à les avoir toutes effacées.

3-il va de soi que l'écran est mis à jour après chaque entrée.

→

```
question(*di)
   -ss_lis(*1)
   -question(*1,*di,o).
question(*1.*2,*di,o)
   -eg(*1,ou(*lis))
   -//
   -af_item
   -window(4,18,1,54)
   -ln
   -i_v
   -outm("        *** lequel des signes suivant : ")
   -ln
   -outm(" taper son no,ou bien 0 si aucun,ou bien t si tous")
   -ln
   -v_n
   -af_lis(*lis,0)
   -i_v
   -ln
   -outm("     est-il present ? ***")
   -v_n
   -ln
   -lirte(*rep)
   -lequel(*lis,*rep,*r)
   -question(*2,*di,*r).
question(*1.*2,*di,o)
   -af_item
   -window(4,18,1,54)
   -ln
   -i_v
   -outm(" ***         le signe ")
   -ln
   -v_n
   -outt(*1)
   -i_v
   -ln
   -outm("             est-il present **")
   -ln
   -v_n
   -ln
   -lirte(*rep)
   -item_plus(*rep,*1)
   -question(*2,*di,*rep).
question(*1,*2,n)
   -window(23,24,32,60)
   -outt(*2)
   -window(4,18,1,54)
   -cls
   -ef_paq(ss_lis).
question("O",*di,o)
   -af_item
   -ln
   -window(21,22,32,60)
   -outt(*di)
   -ef_paq(ss_lis)
   -//.

lequel(*lis,0,n)
   -item_plus(n,ou(*lis))
   -//.
lequel("O",t,o)
   -//.
lequel(*1.*2,t,o)
   -item_plus(o,*1)
   -lequel(*2,t,o).
lequel(*lis,*rep,o)
   -dif(*rep,t)
   -moins(*rep,1,*no)
   -term_plus(*lis,*no,0).

item_plus(n,*1)
   -item_non(*2)
   -ef_paq(item_non)
   -ajtb(item_non(*1.*2)."O").
item_plus(o,*1)
   -item(*2)
   -ef_paq(item)
   -ajtb(item(*1.*2)."O").

item_non("O").

item_ds("O",*1)
   -//
   -impasse.
item_ds(*item.*1,*2)
   -dans_ou(*item,*2)
   -//.
item_ds(*1.*2,*3)
   -item_ds(*2,*3).
```

La forme des clauses est assez rudimentaire de la forme "et" ou "ou". Mais l'introduction de formes plus complexes (liste d'items exclusifs dans une liste, ex: diag(A,ou(1.2.3."O").4.5.6."C") n'est pas spécialement délicate à mettre au point.

Les seules primitives sont rares dans ce programme. Il s'agit de:

- ajtb ajoute en bas d'un paquet de clause.
- cons construit une liste avec des termes.
- esc, cls, ln, outm, outt, outc, prompt sont des procédures d'affichage d'écran.
- ef_paq efface le paquet de clause.
- lirte lis un terme au clavier (terminé par ";").
- // le cut de prolog.
- plus procédure d'addition.

Mais ce qui est totalement transparent, c'est l'utilisation du moteur d'inférence, avec ses boucles récursives qui permettent le travail sur les quelques listes utilisées.

Inconvénients, une certaine lenteur, une place mémoire rédibitoire. Exemple: avec ce programme très simple et 256k de mémoire, il n'est pratiquement pas utilisable, sauf à titre exemplaire !!

---

```
** PROLOG/P  CRIL-CNET  v2.00 - 30 jun 85 - version MS-DOS
** Licence CRIL-CNET no 83131 - Copyright CRIL
** prelude utilise : °-§°nom§.§?-I
```

→

```
wind_end
   -esc
   -outc(46).

window(*1,*2,*3,*4)
   -esc
   -outc(44)
   -plus(*1,31,*x1)
   -outc(*x1)
   -plus(*2,31,*x2)
   -outc(*x2)
   -plus(*3,31,*y1)
   -outc(*y1)
   -plus(*4,31,*y2)
   -outc(*y2)
   -cls.

outl(*1.*2)
   -outt(*1)
   -ln
   -outl(*2).
outl("0")
   -//.

af_item
   -i_v
   -window(5,20,55,79)
   -outm("** ITEMS VALIDES **")
   -ln
   -v_n
   -ln
   -item(*1)
   -outl(*1).

i_v
   -esc
   -outc(112).

v_n
   -esc
   -outc(113).

fin_ses
   -v_n
   -ln
   -outm("  ***** Encore ??? (o ou n) *****")
   -lirte(*1)
   -wind_end
   -cls
   -boucle(*1).

init_ecr
   -window(19,24,1,31)
   -outm(" **** DIAGNOSTIC POSSIBLE : ")
   -ln
   -ln
   -outm(" **** DIAGNOSTIC CONFIRME : ")
   -ln
   -ln
   -outm(" **** DIAGNOSTIC EXCLU : ").

non(*1)
   -*1
   -//
   -impasse.
non(*1).

dans_ou(*1,*1.*2)
   -//.
dans_ou(*1,*2.*3)
   -eg(ou(*4),*2)
   -dans(*1,*4)
   -//.
dans_ou(*1,*2.*3)
   -eg(ou(*4),*1)
   -dans(*2,*4)
   -//.
dans_ou(*1,*2.*3)
   -dans_ou(*1,*3).

ok
   -outm("ok").
```

```
af_lis("0",#no)
    -//.
af_lis(#1.#2,#no)
    -plus(#no,1,#no1)
    -outt(#no1)
    -outm(" - ")
    -outt(#1)
    -ln
    -af_lis(#2,#no1).

term_plus(#1.#2,#no,#cpt)
    -eg(#no,#cpt)
    -item_plus(o,#1)
    -//.
term_plus(#1.#2,#no,#cpt)
    -plus(#cpt,1,#cpt1)
    -term_plus(#2,#no,#cpt1).

boucle(n)
    -//.
boucle(o)
    -chercher.


diag(1,a.d.f.g."0").

START
    -impasse.
```

# *FORTH* Programmation des piles et des registres

par Mr KERJEAN

Le problème se pose de deux façons voisines

1) Programmer un automate à pile ( FORTH par exemple )
2) Progammer une machine à nombre de registres limités.

Ceci en respectant les règles de la programmation:

1) Comprendre ce que l'on fait.
2) Ne pas faire d'opération compliquée,source d'erreur.
3) Faire une documentation claire et simple, en vue de la maintenance future. Tout doit être écrit, c'est à dire:

- les spécifications: but à atteindre,
- la méthode de résolution,
- l'architecture et l'analyse du programme.

4) Faire des programmes faciles à comprendre à la relecture.

Dans le cas présent, les fils directeurs sont:

- créer une série de variables temporaires,
- faire des symboles courts, mais parfaitement définis.

Une telle variable est définie par: son nom, sa localisation, les instants de création, initialisation, lecture, modification et suppression.

Le mieux est de donner des exemples.

NOTATIONS
---------

L'encombrement désignera le nombre de mémoires utilisées à chaque étape du calcul.

Les mots FORTH seront symbolisés comme suit:

| | | |
|---|---|---|
| ↑ DUP | ↓ DROP | ?( IF |
| ↑ OVER | ↕ SWAP | () ELSE |
| P PICK | ↻ ROT | )— THEN |

UTILITAIRES
-----------

Les primitives suivantes se sont avérées à l' usage très commodes:

```
: R! DUP  R ;
VARIABLE M : M! DUP M ! ; : M@ M @ ; : >M M ! ;
VARIABLE N : N! DUP N ! ; : N@ N @ ; : >N N ! ;
```

1) <u>FORMULE DEVELOPPEE</u>

La formule    S = 2 * ( a*b + b*c + c*a )
peut être résulue simplement en déclarant les variables:  VARIABLE a VARIABLE b VARIABLE c VARIABLE S

Mais, surtout si elles sont nombreuses, et s'il faut des impressions de mise au point, il est préférable de les mettre en tableau, surtout que les déclarations correspondantes pourront alors être mises dans un dictionnaire auxiliaire, qui pourra être supprimé à la fin.

→

Le programme devient alors:

```
VARIABLE M 8 ALLOT
: TAB 1- 2 * M + CONSTANT ;
1 TAB a 2 TAB b 3 TAB c 4 TAB S
( a b c -- S )   c ! b ! a !
                 a @ b @ *
                 b @ c @ * +
                 c @ a @ * + 2 * S ! S @
```

A chaque instant, le tableau peut être imprimé par :

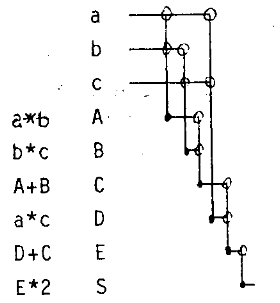
```
: .T M DUP 8 + SWAP  DO I @ . 2 +LOOP ;
```

2) FORMULE CONDENSEE

Cette méthode nécessite de visualiser les différents états de la pile ou des régistres.

Une même formule peut être mise sous plusieurs formes, correspondant chacune à un programme: la surface d'un parallélépipède peut s'écrire, en marquant d'un point les variables utilisées pour la dernière fois:

1 ) S = ( a*b + ḃ*c + ċ*ȧ ) * 2




3443211 encombrement

```
a  b  Ab  cBc cDE2S        pile
↑ 2P *@ 3P *+↑U*↑2*        instructions ( 15 )
a  b A   c BC    E S       commentaires
```

Les mises en facteur sont toujours conseillées, pour réduire les additions, multiplications et surtout divisions. Les variables dont les diagrammes d'utilisation sont imbriqués peuvent être empilées, par exemple sur la pile retour, qui doit être nettoyée a la fin: attention aux branches conditionnelles!

2 ) S = ( a*b + ( ȧ+ḃ ) * ċ ) * 2

a
b
c
A
B
C
D
S

```
   a
  b A/
a
b      B A  D
c        C  D S
↑↑*>R+*R> +↑+     instructions( 10 )
abA  BDA   S      commentaires
R    A  /         pile retour
```

DIVISION PLANCHER

Il s'agit de faire un programme portable de division plancher, qui donne les résultats suivants:

| A | = B | * Q | + R |
|---|---|---|---|
| 8 | 7 | 1 | 1 |
| -8 | 7 | -2 | 6 |
| 8 | -7 | -2 | -6 |
| -8 | -7 | 1 | -1 |

à partir des séquences suivantes, supposées portables:

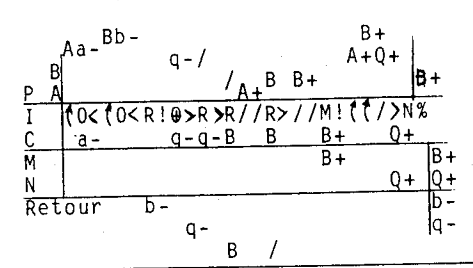
A —s→ a- ; A —//→ A+ ; B —s→ b- ; B —//→ B+ ;;
A+,B+ —/→ Q+ ; A+,B+ —%→ R+ ; a-,b- —⊕→ q-;;
q- si ( Q+ 1+ ± Q ; B+ R+ R1 ; ) ;;
b- si ( R1 ± R ; ) ;;

B
b-
B+
A
a-
A+
Q+
R+
q-
Q
R1
R

s :signe - // : valeur absolue
/ :division % : reste
± :NEGATE ⊕ : OU exclusif
, : séparateur de variable
; : d'instruction
;; : de séquence

En programmation parallèle, les instructions peuvent être exécutées simultanément, en programmation séquentielle, l'ordre des instructions est quelconque, dans une même séquence.

23344555532 encombrement




```
    q-  B+  Q+  Q    b-   Q
PR+         R1          R
I   R> ?(M@↑-N@1+±>N)-R> ?(±)-N
C   q-  B+  Q+       b-   Q
MB+
NQ+
Rb-                  /
 q-/
```

Dans le cas d'une machine à registres, en comptant le nombre d'utilisation de chacun d'eux, il est possible de faire l'affectation suivante:

| Registre | Variables | Utilisation |
|---|---|---|
| R1 | A a- q- R1 | 8 |
| R2 | B b- | 5 |
| R3 | A+ R+ | 5 |
| R4 | B+ | 4 |
| R5 | Q+ Q | 3 |

Il faut toutefois noter que:
- il existe un programme plus simple
- une telle étude ne se justifie que pour un programme d'usage très fréquent, il faut alors bien le documenter. En général, il vaut mieux mettre l'accent sur la lisibilité plus que sur la rapidité ou l'encombrement.